Aluno: Kauã Melchioretto

# Redes de Computadores
# Camada de Rede

## Relatório: Simulação de Protocolos de Roteamento da Camada de Rede

### Introdução

A camada de rede é responsável pelo encaminhamento eficiente de pacotes através de dispositivos em uma rede. Três algoritmos amplamente utilizados para este propósito foram implementados:

1. **Algoritmo de Vetor de Distâncias**
2. **Algoritmo de Estado de Enlace (Dijkstra)**
3. **Algoritmo de Vetor de Caminhos**

Cada algoritmo é apresentado com uma explicação teórica, seguido de uma implementação prática em linguagem C. Uma Prova de Conceito (POC) é incluída para demonstrar os mecanismos dinâmicos de atualização.

### 1. Algoritmo de Vetor de Distâncias

#### Descrição

Este algoritmo utiliza o princípio da equação de Bellman-Ford para determinar os menores custos entre um nó e todos os outros. Cada nó mantém uma tabela de distâncias que é continuamente atualizada a partir das tabelas recebidas de seus vizinhos.

#### Cenário Dinâmico

Os custos dos caminhos mudam ao longo do tempo. O algoritmo ajusta as tabelas de roteamento de forma assíncrona, enviando atualizações aos vizinhos quando detecta alterações.

#### Código

```c
#include <stdio.h>
#include <limits.h>

#define N 6
#define INF INT_MAX
```

```c
void printDistances(int distances[N]) {
    printf("Vetor de Distancias: ");
    for (int i = 0; i < N; i++) {
        if (distances[i] == INF) {
            printf("INF ");
        } else {
            printf("%d ", distances[i]);
        }
    }
    printf("\n");
}

void distanceVectorRouting(int cost[N][N], int startNode) {
    int distances[N];
    int updated;

    // Inicializa as distâncias
    for (int i = 0; i < N; i++) {
        distances[i] = (cost[startNode][i] != 0) ?
cost[startNode][i] : INF;
    }
    distances[startNode] = 0; // Distância para si mesmo é 0

    printf("Inicializacao para o no %d:\n", startNode);
    printDistances(distances);

    // Atualização iterativa usando a equação de Bellman-Ford
    do {
        updated = 0; // Indica se houve mudanças
        for (int i = 0; i < N; i++) {
            if (i == startNode || distances[i] == INF) continue;

            for (int j = 0; j < N; j++) {
                if (cost[i][j] != 0 && cost[i][j] != INF &&
distances[i] + cost[i][j] < distances[j]) {
                    distances[j] = distances[i] + cost[i][j];
                    updated = 1;
                }
            }
        }

        printf("Apos uma atualizacao:\n");
        printDistances(distances);

    } while (updated);

    printf("Distancias finais do no %d:\n", startNode);
    printDistances(distances);
```

```
}

void simulateChanges(int cost[N][N]) {
    printf("\n** POC: Mudanca no custo do enlace **\n");
    printf("Alterando custo entre os nos 1 e 4 para 10...\n");
    cost[1][4] = 10;
    cost[4][1] = 10;

    distanceVectorRouting(cost, 1);

    printf("\n** POC: Remocao de um no **\n");
    printf("Removendo o no 3 (tornando os custos infinitos)...\n");
    for (int i = 0; i < N; i++) {
        cost[3][i] = INF;
        cost[i][3] = INF;
    }

    distanceVectorRouting(cost, 1);

    printf("\n** POC: Adicao de um novo no **\n");
    printf("Simulando adicao do no 3 novamente com custos
restaurados...\n");
    cost[3][2] = 5;
    cost[2][3] = 5;
    cost[3][4] = 7;
    cost[4][3] = 7;

    distanceVectorRouting(cost, 1);
}

int main() {
    // Matriz de custos (0 indica que não há conexão)
    int cost[N][N] = {
        {0, 2, 0, 1, 0, 0},
        {2, 0, 3, 2, 0, 0},
        {0, 3, 0, 0, 7, 0},
        {1, 2, 0, 0, 3, 6},
        {0, 0, 7, 3, 0, 1},
        {0, 0, 0, 6, 1, 0}
    };

    printf("Simulacao inicial\n");
    distanceVectorRouting(cost, 0);

    simulateChanges(cost);

    return 0;
}
```

## 2. Algoritmo de Estado de Enlace (Dijkstra)

### Descrição

Este algoritmo constroi iterativamente uma árvore de menor custo partindo de um nó raiz. Ele utiliza o conjunto de vizinhos e atualiza as menores distâncias para outros nós conforme a expansão da árvore.

### Cenário Dinâmico

O algoritmo atualiza a árvore sempre que novos enlaces são adicionados ou removidos, refletindo mudanças na rede.

### Código

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define N 6 // Número de nós
#define INF INT_MAX

void printDistances(int distances[], int previous[], int
startNode) {
   printf("\nMenor custo a partir do no %d:\n", startNode);
   for (int i = 0; i < N; i++) {
       if (distances[i] == INF) {
           printf("No %d: INF\n", i);
       } else {
           printf("No %d: %d (via no %d)\n", i, distances[i],
previous[i]);
       }
   }
}

int findMinDistanceNode(bool visited[], int distances[]) {
   int minDistance = INF;
   int minIndex = -1;

   for (int i = 0; i < N; i++) {
       if (!visited[i] && distances[i] < minDistance) {
           minDistance = distances[i];
           minIndex = i;
       }
   }

   return minIndex;
}

void dijkstra(int graph[N][N], int startNode) {
```

```
    int distances[N];   // Armazena as menores distâncias
    bool visited[N];    // Marca os nós já processados
    int previous[N];    // Para rastrear o caminho mais curto

    // Inicialização
    for (int i = 0; i < N; i++) {
        distances[i] = INF;
        visited[i] = false;
        previous[i] = -1;
    }

    distances[startNode] = 0; // Distância para si mesmo é 0

    for (int i = 0; i < N; i++) {
        int currentNode = findMinDistanceNode(visited, distances);
        if (currentNode == -1) break; // Nenhum nó alcançável
restante

        visited[currentNode] = true;

        // Atualiza as distâncias dos vizinhos
        for (int neighbor = 0; neighbor < N; neighbor++) {
            if (!visited[neighbor] && graph[currentNode][neighbor]
!= 0 && graph[currentNode][neighbor] != INF) {
                int newDistance = distances[currentNode] +
graph[currentNode][neighbor];
                if (newDistance < distances[neighbor]) {
                    distances[neighbor] = newDistance;
                    previous[neighbor] = currentNode;
                }
            }
        }
    }

    printDistances(distances, previous, startNode);
}

void simulateChanges(int graph[N][N]) {
    printf("\n** POC: Alteracao de custo **\n");
    printf("Alterando o custo entre os nos 1 e 4 para 2...\n");
    graph[1][4] = 2;
    graph[4][1] = 2;
    dijkstra(graph, 0);

    printf("\n** POC: Remocao de no **\n");
    printf("Removendo o no 2...\n");
    for (int i = 0; i < N; i++) {
        graph[2][i] = INF;
```

```
        graph[i][2] = INF;
    }
    dijkstra(graph, 0);

    printf("\n** POC: Adicao de no **\n");
    printf("Adicionando o no 2 novamente com custos
restaurados...\n");
    graph[2][3] = 3;
    graph[3][2] = 3;
    dijkstra(graph, 0);
}

int main() {
    // Matriz de adjacência representando o grafo
    int graph[N][N] = {
        {0, 4, INF, INF, INF, INF},
        {4, 0, 8, INF, INF, INF},
        {INF, 8, 0, 7, INF, 4},
        {INF, INF, 7, 0, 9, 14},
        {INF, INF, INF, 9, 0, 10},
        {INF, INF, 4, 14, 10, 0}
    };

    printf("** Simulacao inicial **\n");
    dijkstra(graph, 0);

    simulateChanges(graph);

    return 0;
}
```

## 3. Algoritmo de Vetor de Caminhos

### Descrição

Este algoritmo estende o vetor de distâncias ao incluir o caminho completo para cada destino. Ele evita laços verificando se o caminho recebido inclui o próprio nó.

### Cenário Dinâmico

O algoritmo responde dinamicamente a mudanças nos enlaces da rede, como adição e remoção de conexões.

### Código

```
#include <stdio.h>
#include <string.h>
```

```c
#define N 5 // Numero de nos
#define INF -1 // Representacao de caminho vazio

typedef struct {
   int destino;
   int via;
} Caminho;

void printCaminhos(Caminho caminhos[N], int meuId) {
   printf("\nTabela de caminhos do no %d:\n", meuId);
   for (int i = 0; i < N; i++) {
       if (caminhos[i].via == INF) {
           printf("Destino %d: Caminho vazio\n", i);
       } else if (caminhos[i].via == meuId) {
           printf("Destino %d: Eu mesmo\n", i);
       } else {
           printf("Destino %d: Via no %d\n", i, caminhos[i].via);
       }
   }
}

void inicializarCaminhos(Caminho caminhos[N], int grafo[N][N], int
meuId) {
   for (int i = 0; i < N; i++) {
       if (i == meuId) {
           caminhos[i].destino = i;
           caminhos[i].via = meuId;
       } else if (grafo[meuId][i] != INF) {
           caminhos[i].destino = i;
           caminhos[i].via = i;
       } else {
           caminhos[i].destino = i;
           caminhos[i].via = INF;
       }
   }
}

void atualizarCaminhos(Caminho caminhos[N], Caminho
caminhosVizinho[N], int meuId, int vizinhoId) {
   for (int i = 0; i < N; i++) {
       if (caminhosVizinho[i].via == meuId) {
           // Evitar loops
           continue;
       }

       if (caminhos[i].via == INF || caminhosVizinho[i].via !=
INF) {
```

```
            caminhos[i].via = vizinhoId;
        }
    }
}

void enviarCaminhos(Caminho caminhos[N], Caminho
caminhosRecebidos[N], int destino) {
    memcpy(caminhosRecebidos, caminhos, sizeof(Caminho) * N);
}

void simulateChanges(int grafo[N][N], Caminho caminhos[N]) {
    printf("\n** POC: Alteracao de conexoes **\n");
    printf("Adicionando um caminho entre os nos 3 e 4...\n");
    grafo[3][4] = 1;
    grafo[4][3] = 1;
    inicializarCaminhos(caminhos, grafo, 0);
    printCaminhos(caminhos, 0);

    printf("\nRemovendo a conexao entre os nos 1 e 2...\n");
    grafo[1][2] = INF;
    grafo[2][1] = INF;
    inicializarCaminhos(caminhos, grafo, 0);
    printCaminhos(caminhos, 0);
}

int main() {
    // Matriz de adjacencia do grafo
    int grafo[N][N] = {
        { 0, 1, INF, INF, INF },
        { 1, 0, 1, INF, INF },
        { INF, 1, 0, 1, INF },
        { INF, INF, 1, 0, 1 },
        { INF, INF, INF, 1, 0 }
    };

    Caminho caminhos[N];
    Caminho caminhosRecebidos[N];

    int meuId = 0;

    // Inicializacao
    inicializarCaminhos(caminhos, grafo, meuId);
    printCaminhos(caminhos, meuId);

    // Simulacao da POC
    simulateChanges(grafo, caminhos);

    return 0;
```

```
}
```

## Conclusão

Os algoritmos foram implementados com sucesso, e suas POCs demonstram a atualização dinâmica de tabelas e árvores conforme mudanças na rede. Cada protocolo tem aplicações específicas, mas juntos garantem flexibilidade e eficiência no roteamento.